The 13th Technology of Deep Space One[1]

Nicolas F. Rouquette, Tracy Neilson, George Chen
Jet Propulsion Laboratory
Pasadena, CA 91109
(818) 354-9600
{nicolas.rouquette, tracy.neilson, george.chen}@jpl.nasa.gov

Abstract—This paper describes an innovative approach to spacecraft fault protection based on automatic code-generation techniques.

TABLE OF CONTENTS

technology validation

1. INTRODUCTION

On October 24th, 1998, the Deep Space One (DS-1) spacecraft launched aboard a Delta II rocket as the first step towards the bold task of testing and validating 12 new technologies for future missions. This launch also represented yet another thrilling event; namely, the successful test and validation of a 13th heretofore undisclosed technology: model-based code-generation of the spacecraft's system-level fault-protection (FP) software from behavioral state diagrams and structural models.

Until DS-1, the Jet Propulsion Laboratory (JPL) had not used code-generation techniques on large scale for avionics software. However, the constraints of the mission design and development cycle, limited budget and resources dictated a departure from past practices. The analysis of the system-level issues started in March 1997 with minimal staff while the actual design and development of the fault-protection engine started in earnest in June 1997. Radical departures from past projects were necessary to complete the design, development and testing of the system-level fault-protection in time for launch. The requirement that post-launch activities be directed by fault protection further increased the difficulty of the task; on other spacecraft, such activities are typically handled with sequences.

First, a decision was made in June to use the successful Mars Pathfinder (MPF) fault-protection engine because this system made a nice separation of the various concerns between detecting faults, signaling faults and executing fault responses. However, the limited resources available precluded a duplication of MPF's design and development approach because we had too few software engineers and too much uncertainty about the hardware, the flight software and the scope of the system-level issues. This high degree of uncertainty translates into a high degree of design instability and volatility. To accommodate this difficult state of affairs, we shifted the design and development of the system-level fault-protection software from low-level concerns about the C language to higher-level concerns about system-level requirements, issues, strategy, and tradeoffs. To make this top-down design approach work in a team environment while retaining sufficient implementation flexibility, we standardized on using state diagrams and attribute specifications as design notations for describing the behavior and structure of fault-protection designs.

In this paper, we describe the process we used to leverage model-based code generation from state diagrams and structural specifications to better respond to the evolving requirements and scope of DS-1's system-level fault-protection design, development, test and operation. The evolution of the high-level design and the low-level changes in the flight software architecture and interfaces contributed to multiplying the number and frequency of fault-protection software releases thereby creating a multitude of software integration issues. To address the resulting software integration issues, we broadened the scope of code generation to other forms of model-based analysis techniques more traditionally associated with first-principle's reasoning about physical models. Additionally, we describe our in-flight launch and initial acquisition experience.

2. MOTIVATION

In 1997, the schedule to complete within 12 months the design, implementation and testing of the FP software for DS1 looked challenging. At that time, launch was scheduled for July 1998; it was later delayed due to late hardware deliveries. To be sure, the spacecraft is already complex due to a single-string design, tight pointing requirements and 12 new hardware and software technologies. Strategically, we decided to apply code-generation techniques for several reasons:

Precedent: There was a precedent on DS-1 for the code generation of the monitors of the Remote Agent Experiment [2].

Schedule: There were too few software engineers available to enable quick turnarounds from design to code.

Stability & flexibility: Changes to the software interfaces and architecture were expected. This in turn threatened

[1] 0-1234-5678-0/99/$5.00 © 1999 IEEE

to further exacerbate FP software issues.

Priorities: Strategically, it made sense to focus the team efforts on analyzing the intricate complexities of the system-level interactions instead of dividing software-engineering tasks across the team.

Reviewable functionality: The complexity of the spacecraft made analyzing scenarios a difficult and challenging task. We needed a design notation sufficiently clear to allow several people to follow an analysis discourse and sufficiently compact to facilitate such reviews.

Code-generation techniques allowed us to make an important separation of concerns between what system-level FP should do (design) and how it should do it (implementation).

3. Design Strategy

Wary of the pitfalls of code generation, we emphasized a rigorous review process to track progress, to identify technical difficulties, and to calibrate the scope of the fault protection design to fit within our resources.

At the level of the FP team, we engaged on a series of semi-weekly meetings to debate the FP issues, to review supporting materials and to discuss FP designs. Each FP monitor and response was managed under the cognizance of an engineer responsible for the two main aspects of the design: structure and behavior. Since the scope of the FP design was still in flux, it was more important to focus on this primary design content rather than include secondary design aspects such as telemetry, commanding, and test interfaces. From a software viewpoint, we counted on the software architecture to handle all secondary design concerns in a standard and consistent manner. Thus, the bulk of the design meetings focused instead on the primary components of the fault-protection design (see Fig. 1), namely, a structural definition:

- a software interface with the flight software manager task that calls the monitor;
- internal variables to hold state information and intermediate computations;
- parameters to calibrate and control the monitor or response behavior;

and a behavioral definition:

- a state diagram defining the behavior of the monitor or response as a function of its inputs, internal variables and previous state.

Building from prior experience with high-level specification languages [2] and high-level diagrammatic notations for behavior, we concluded that reviewable functionality was the most important criteria to enable the kind of high-level peer design reviews necessary to meet our schedule.

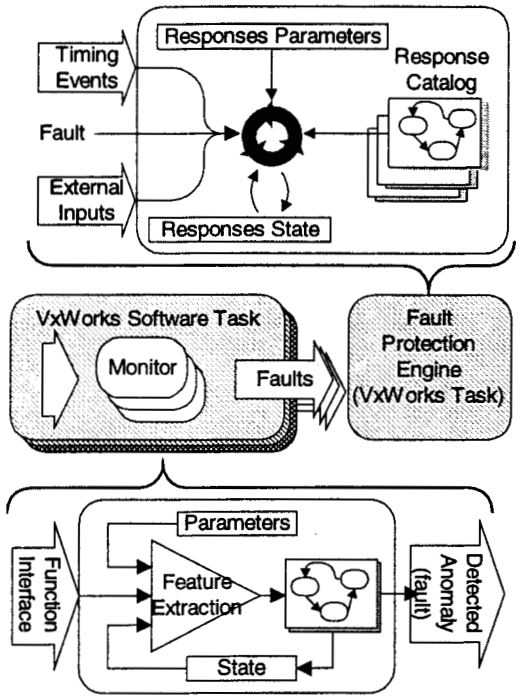


Figure 1 The DS1 Fault-Protection architecture

3.1 Statecharts

Statecharts provide a good mix of compactness, semantic precision and readability for representing behavior. These attributes are important to minimize misunderstandings, to conduct effective design reviews and to disseminate the most important details of the FP design to a broad audience of various backgrounds. Despite the natural fit of state diagrams as a compact and precise high-level design notation, several practical issues remain. Specifically: 1) establishing a standard diagrammatic notation, 2) defining semantics suitable for the project needs and 3) translating statechart designs into flight software (See § 4.2).

Figure 2 shows a simplified view of the DS1 statechart controlling the system-level activities following separation from the launch vehicle. For DS1, we used the Matlab Stateflow® toolbox to design such statecharts. This tool enforces standard diagrammatic conventions for representing statecharts. The topology of the launch statechart is a three-level hierarchy. At the top level, there are two states, 'init' and "launch". At a given level, a transition from a black dot indicates the starting state for that level. The transition from the 'init' state is predicated on the 'LAUNCH' event. The fault-protection engine broadcasts this event when it receives indication that the spacecraft has booted-up and separated from the launch vehicle. This figure also illustrates some of the extensions to the statechart notation we introduced for DS1. For example, we introduced the notion of 'GOTO' either to show explicitly the notion of statechart reuse like a sub-routine function (e.g., the 'detumble' statechart is used elsewhere), or to modularize a complex process into smaller

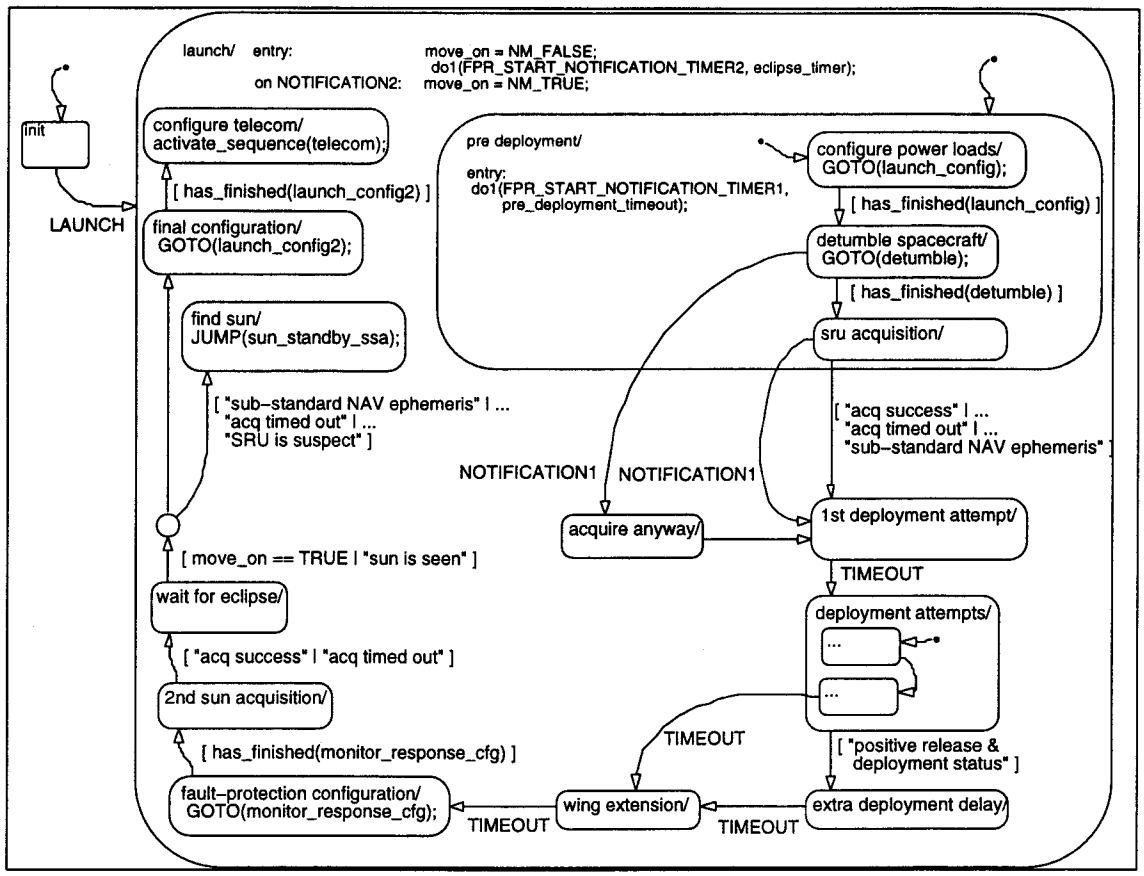


Figure 2 Simplified Launch Statechart

chunks (e.g., the 'launch_config' statechart is only used for launch processing).

The combination of a high-level design approach with code generation of low-level software implementation resulted in a strategy that:

- Facilitates frequent tuning of the overall FP design;
- Enables implementation flexibility to adapt to changing interfaces and software requirements;
- Minimizes the software coding effort to concentrate instead on design issues;
- Reduces the number of errors introduced by the coding process.

The design strategy outlined placed great emphasis on statecharts. This approach matched very well with the needs of spacecraft fault protection because the essence of failure recovery is intrinsically a behavioral problem. In our approach, we relied on the power of teamwork to enact the best possible behavioral strategies to address the failure recovery needs of DS1.

However, this behavioral approach to fault protection design necessitated an efficient, streamlined design process to analyze, review and test failure scenarios against the fault protection design. To meet this challenge, we made a strategic decision to automate as much as possible the time-consuming and error-prone aspects of the process through software code-generation technology as described next.

4. Code Generation

4.1 Code Generation Issues

Today, many software-engineering design tools feature code generators that produce target-specific software from high-level design information. Code generators are highly popular; they enforce a rigorous approach to generating software due to their systematic and consistent operation. A general-purpose code-generation technology does not exist since there is no general-purpose mechanism to understand arbitrary design information. Given a high-level design language and an associated code-generator capable of handling design models written in that language, we can establish a taxonomy of code-generation properties to compare the available technologies. The dimensions of our code-generation taxonomy derive from five pragmatic issues as described below.

Algorithmic Customization

Are the algorithm details of the code-generation approach accessible and modifiable?

Perhaps the most important risk factors involve limitations in output and algorithmic customization. Changes to the

project environment and requirements may call for unprecedented flexibility in code-generation algorithms and output. If this flexibility is at a level beyond that which the code-generation technology provides, it becomes necessary to devote resources to work around existing code-generation limitations. This issue becomes of central importance in generating code from statecharts, particularly when the semantics of the statechart notation are adapted to match the project's architecture and requirements.

Without adequate customization potential, it may be necessary to engineer a new code-generator to gain sufficient control over the code-generation process. Alternatively, it is possible to post-process the output of the code generator. Either way, the sudden increase of resources devoted to supporting or working around the existing code-generation tool instead of what the tool generates can result in unacceptable scheduling delays and expansive costs overruns. These undesirable effects are easily attributable to a code-generation technology that, a-priori, might have matched the project needs but doesn't a-posteriori. Drastic decisions may follow such as switching tools, or abandoning altogether the code-generation approach to revert to a more conventional and predictable software engineering process.

A-priori knowledge

What knowledge is embedded in the code generator?

Code generation algorithms are often embedded inside a code generator. While software vendors typically hide such algorithms inside their code-generation products, such practices can make customizing a code generator to fit a project's needs a difficult task. In fact, the practice of hiding key algorithms often results in users having to adopt a vendor's viewpoint on code generation. This invariably occurs at the expense of the project, which then needs to adapt its processes and organizations to accommodate the product's constraints instead of adapting the product to the project's constraints. The former approach places a burden on the project to be sufficiently flexible or conservative enough to avoid reaching beyond the capabilities a code generation tool can reasonably support. This issue is at the heart of many project failures.

It is often difficult to cleanly separate all knowledge about a domain from the code generation mechanisms. Commercial tool vendors often hide strategic technologies inside their code generators. This unfortunately results in sealing domain-specific knowledge about what is being generated inside the code generator itself. Without access to this domain knowledge, it is often difficult, and sometimes legally impossible, to customize the domain knowledge to fit the project needs without building a code generator from scratch. To address this issue, many code generators use a template approach to encapsulate all domain-specific knowledge in a customizable form. The idea of a template is to describe what the generated code looks like in terms of boilerplate text annotated with special tags. A scripting language defines a framework to replace tags with results computed from the input data to the code generator as illustrated in Fig. 3. This figure illustrates a simple template mechanism where the code generator replaces tags in the template, `@<name>` and `@<date>`, with their corresponding values computed from the available domain information. The literal text, combined with the text produced from evaluating the tag, constitutes the generated output.

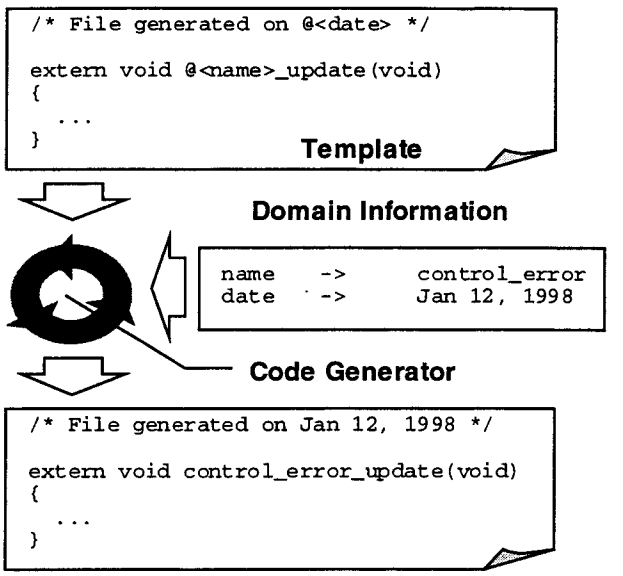


Figure 3 Code generation from templates.

Output Customization

What flexibility the code generation framework provides for custom code?

Code generation tools typically provide output customization mechanisms that fall in three categories: 1) code generation templates as described earlier, 2) flags that control the form of specific output elements and 3) protected regions to insert and preserve user-provided text in the generated code.

The two code-generators used for DS1 fault protection relied on the template mechanism to customize the code generator output. The second approach assumes that the customization flags are sufficiently expressive to anticipate all possible needs. The protected region approach complements the customization flag approach: users can add arbitrary code to the generated output at designated areas of the generated output. This customization practice then leads to a phenomenon of round-trip software engineering where user-defined customizations are preserved across multiple code generation passes. While this is appropriate when code generation is not a complete solution, we aimed on DS1 to generate all aspects of the software including structure and behavior since we didn't have the resources to perform round-trip engineering.

Approach

How does the code generator work?

Rodney Bell [1] made a distinction between three approaches: structural, behavioral and translative. Each approach makes assumptions about the content of the model and therefore embeds a-priori knowledge about how the model information is organized. The structural approach views model information as characterizing the nature of the objects in the model and relations such objects have. The generated code then corresponds to the structural declarations and definitions of such objects and relations in the target language. The behavioral approach views model information as characterizing the internal behavior of objects, typically expressed in terms of state diagrams. The generated code then corresponds to the implementation of such state machines into the target language. Distinguishing between an application domain and architecture for this domain, the translative approach generates code according to a mapping between the application domain and the architecture.

The template approach uses a set of boilerplates, each corresponding to a type of desired output. Each such boilerplate also contains special macro annotations that the code generator replaces with information from the input design semantic nature of the annotation. Each generated file thus corresponds to an expanded boilerplate where all macros annotations have been replaced in this manner.

The translative approach is not often used; the code generation process is driven from the mapping of domain objects onto architecture. In this approach, elements of both the domain and of the architecture are inputs to the code generation process.

DS1 used a template approach, which we describe further in Sec. 4.2.

Risk factors

What impact code generation has on cost, schedule and technical risks?

For DS1 fault protection, we chose a code-generation approach that espoused a zero customization policy towards the generated software. While regarded as a sign of flexibility, the ability to add custom code significantly increased the complexity of a software engineering process based on code generation. Custom code becomes one more source of information that needed to be managed. This approach also introduced further complexity because it is an unnecessary degree of freedom: Solving issues in code generation at the level of the code generation inputs or even the algorithm yields architecturally elegant and consistent solutions across the board. Custom code stands out as an exception that further complicates the debate between fixing the architecture, the design and tweaking the result. Eventually, custom code stratifies into layers of customizations that break the consistency of the process and become very difficult to manage. On DS1, there were no provisions for a generic customization capability. This alleviated the need to merge machine and user generated code.

4.2 Code Generation Approach

The Matlab Stateflow® toolbox performs behavioral code generator from statecharts. It is also an example of a template-based code generator where the template annotation and boilerplate code is embedded inside the code generator.

In addition to this toolbox, we used another code generator, a Iterative Template Language Compiler (ITLC). This code generator is inspired from the Matlab Real-Time Workshop® Target Language Compiler (TLC). The Matlab TLC is a powerful mechanism to associate a customizable code-generation algorithm for each block in a Matlab Simulink® model. Simulink models are defined by an interconnected set of function blocks where inter-block connections define the flow of data in the model while blocks define the processes that operate on that data. To provide a flexible code-generation mechanism, the Matlab TLC combines aspects of the Perl string-manipulation capabilities, the tag mechanism of HTML and the general-purpose computing capabilities of Matlab into a code-generation language. Code generation is a self-contained process entirely driven by the properties of each block in a Simulink® model and by the topology of the block interconnections.

The Matlab TLC associates a template to a specific block type. The DS1 ITLC has a coarser granularity as a template represents the specific boilerplate for a given kind of generated file (headers, source files, documentation, test driver, etc...). Figure 4 shows the functional block diagram of the ITLC we designed for DS1.

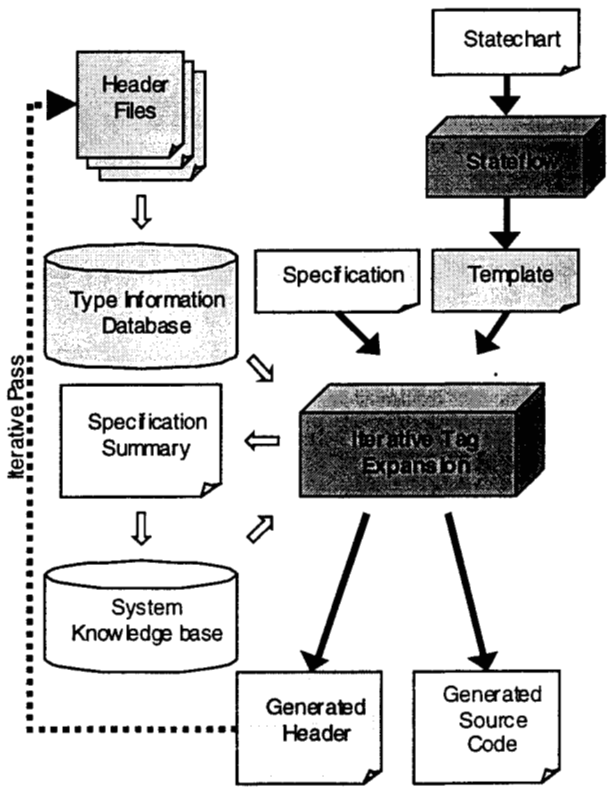


Figure 4: Code generation process (Recursive Template Language Compiler)

Whereas the Matlab TLC operates on a Simulink block and a template for that block type, our ITLC operates on a specification and a template. The set of annotation tags constitutes the semantic link between the template and the specification in that tags can reference elements of the specification. Additionally, the Matlab TLC is a self-contained environment whereas our ITLC taps on other additional sources of information as described in the next sections.

4.3 Knowledge of Software Interfaces

The set of all header files corresponding to the public interfaces in the flight software is available to the code generator in terms of a type information database constructed from such headers. For example, this database provides a convenient mechanism to generate software for the ground system to decode the values of enumerated types into their corresponding symbolic enumerated names. Additionally, this database is recursively updated with header files produced by the code generator. This property enables the code generator to obtain knowledge of the data structures generated thereby simplifying the process of writing templates that refer to other structures previously generated.

To illustrate this process, we will use a simplified specification of an attitude control error monitor as shown here.

```
begin variables
 MDC_States_t acs_mdc
 double error[3]
end variables
```

Templates extract this information and produce header files defining C data structures for the flight software:

```
typedef struct {
#forall variables
  @<variable_type> @<variable_name>;
#end variables
} mon_@<name>_state_t;
```

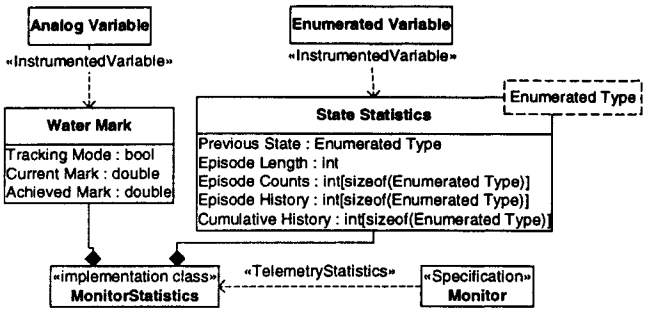
Once instantiated, we obtain a definition that is parsed and added to the type database.

```
typedef struct {
 MDC_States_t acs_mdc;
 double error[3];
} mon_control_error_state_t;
```

This iterative software construction simplifies the process of instrumenting state variables since they are known to the code generator. For example, each fault-protection monitor updates a number of statistical properties about the data measurements it is monitoring such as minimum and maximum values (also known as watermarks). We also defined state statistics for summarizing the behavior of discrete-valued measurements. Thus, a monitor specification defines a set of watermarks and state statistics to compute as illustrated below for the running example:

```
begin statistics
STATE_SUMMARY(mdc_states)
HIGH_WATER_MARK(control_error[3])
end statistics
```

This statistics specification has a meaning in the context of an architecture for computing the statistical properties of state variables like the one illustrated below in UML notation:



Knowledge of what constitutes an analog variable and an enumerated variable stems from analyzing the type of each variable involved. On DS1, the following template illustrates how this process takes place in a 2-step manner:

```
typedef struct {
#forall variables
 @<declare_telemetry(@<variable_name>)>
#end variables
} mon_@<name>_statistics_t;
```

At the first tag expansion pass, each variable name is expanded separately. At the second tag expansion pass, the declaration of telemetry statistics is applied to each variable found in the first pass. If the variable is not instrumented in the specification, nothing is produced; otherwise, the water mark or state statistic schema is applied according to the type of the variable. In the example, this produces the following result assuming that there are 18 possible values for the MDC_States_t enumeration:

```
typedef struct {
 MDC_States_t previous_acs_mdc;
 int acs_mdc_episode_length;
 int acs_mdc_episode_counts[18];
 int acs_mdc_cumulative_counts[18];
 int error_tracking_mode[3];
 double error_current_mark[3];
 double error_achieved_mark[3];
} mon_control_error_statistics_t;
```

This iterative code generation process comes full circle in that the above definition is incorporated in the type information database to support additional code generation about the telemetry statistics. On DS1, we generate from such definitions ground software to decode the telemetry packets received from the spacecraft and print them according to their engineering definitions and units. This process helps us enforce consistency between on-board and ground software as we change the design, add local state variables or change the flight software interfaces.

4.3 Aggregate Summarization of Specifications

The second difference with the Matlab TLC approach stems from the process of summarizing specifications into aggregate specifications. This provides the code generator with a closed-world view of all relevant entities. This knowledge was particularly useful on DS1 to enable the code generation templates to process collections of entities like the set of all monitors or the set of all responses.

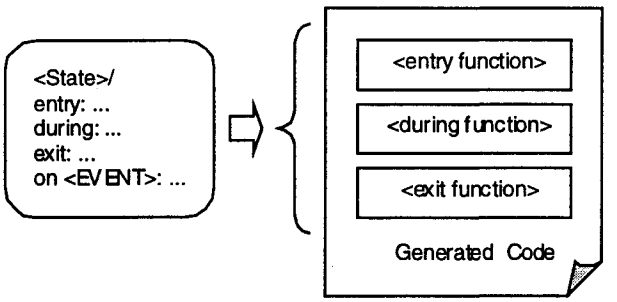
A more subtle application of aggregate summarization occurred in the definition of the fault-protection telemetry to indicate the progress of a fault-response statechart execution. The underlying problem that motivated this effort was the need of recording a sufficient amount of information to trace what actions a fault-response statechart performed during its execution. There is limited memory available to store such information; yet a response can spend a significant amount of time waiting for events to occur (e.g., deploying the panels). We also did not want to overload the statechart notation with superfluous logging commands because they would complicate our design and review process and inconsistencies could emerge between the logging commands and the actual state definitions. To define an automatic statechart execution logging mechanism, we used the aggregate summary of all statecharts to compute the set of all possible statechart execution paths. As shown in Fig. 2, one statechart can call another statechart as a subroutine via the 'GOTO' mechanism. Thus, we can define a statechart call graph from the possible ways 'GOTO' statements can be nested. On DS1, this graph contains 93 vertices and 92 edges. Each vertex represents a distinct path from a top-level statechart to the currently executing statechart through a stack of nested 'GOTO' calls. By monitoring statechart state transitions and 'GOTO' calls, we defined an algorithm to update the current vertex in the statechart call graph. Then, it suffices to record the id of the current vertex in that graph to fully explain the history of statechart executions up to the current statechart state. On the ground side, there is a dual algorithm which looks up the graph vertex ids and produces the corresponding paths of statechart GOTO calls. This combination of graph encoding/decoding algorithms produces a minimum-length mechanism to monitor and record statechart execution on DS1. With this compact representation, we also summarized the set of all relevant external inputs necessary and sufficient to fully explain all statechart transitions based on external events. This example illustrates one of the customizations of the Stateflow® toolbox we performed for DS1; other customizations are described in the next section.

4.4 Extensions and Customizations of Stateflow

Modifying a behavioral code generator like Stateflow raises an acute issue: the vendor, here the Mathworks, has spent considerable effort in validating the code generator. Specifically, the vendor has the responsibility to ensure that the behavioral semantics of the generated code precisely match the semantics of the diagrammatic statechart notation. In modifying the code generator, we introduce the possibility of breaking this property, thereby exposing the project to additional risks and costs. At first approximation, this would preclude modifying the Stateflow code generator. However, it was necessary for the DS1 flight software architecture to address a number of difficult issues:

1) Altering the form and organization of the generated code to comply with the DS-1 programming guidelines.

The Stateflow-generated code is isomorphic to the topology of a statechart in the following sense that each state translates into a fixed set of functions:



Our modification consisted in changing the naming conventions used for the names of the entry, during and exit functions. These naming convention changes were also propagated to relevant areas of the generated code as well (e.g., the enumerated list of state names).

2) Altering the organization of the generated code to preserve a consistent topological order so that version control software can track local changes in the state diagrams to local changes in the generated code.

The order in which Stateflow visits states controls the order in which state functions are generated. Stateflow uses two criteria for visiting states: the hierarchical structure of the statechart and a Matlab-generated identifier. The former controls how hierarchically nested states are visited; the latter controls how sibling states at a given level in the hierarchy are visited. Our modification consisted in using an external criterion for visiting sibling states: we used the alphabetical order of the state names.

3) Defining project-specific semantics for event broadcast in state diagrams.

For each external event defined, Stateflow generates broadcast functions to inform the statechart of the occurrence of such events. However, this broadcast mechanism is unconditional. For DS1, we had to define lexical scoping rules for broadcasting events. The DS1 fault-protection engine allows three concurrent timers for each statechart: TIMEOUT, NOTIFICATION1, and NOTIFICATION2. A special action provides a mechanism to start one of these timers. The fault-protection engine is notified whenever a timer delay expires. However, the fault-protection engine cannot broadcast the corresponding

EVENT without some checking. External inputs may have changed and caused state transitions where the timer is no longer relevant as illustrated in Fig. 2 for the 'deployment attempts' superstate. The '1st deployment attempt' state guarantees a minimum of panel deployment effort. Once this is complete however, we want to make multiple passes at subsequent deployment attempts until there is a positive confirmation of panel deployment from the sensors. Suppose that the positive confirmation occurs before the pending timer delays expire. The transition to 'extra deployment delay' occurs where a new timer delay is requested. There is now a race situation between the expiration of these two timer delays. To handle such situations properly, it is customary to tag timer requests and match them with the tag of the timer expiration notification. However, this does not resolve the issue that the timer can become obsolete because of some other transition occurred. To handle such situations, we tag the timer requests with the id of the state where the timer is requested. Upon receipt of a timer delay expiration, we check that the current state of the statechart is a substate of the tagged timer state. This mechanism makes timer requests lexically scoped relative to the state where the timer delay is requested.

4) Extending Stateflow to external datatypes.

Normally, Stateflow handles all data types that Matlab Simulink knows about. Unfortunately, this does not include C pointers, enumerated types and nested C structures. Since the DS1 software interfaces use these datatypes, we had to perform some post-processing of the Stateflow-generated code to restore the proper pointer and structure references. In that regard, Stateflow-generated files are considered like templates for ITLC. The expansion of such templates then makes extensive use of the type information database and makes the closed-world assumption that all types are defined in the database. This property allows us to define a simple search & replace algorithm: given an architectural definition of which data structures are accessible in a statechart, we can exhaustively enumerate all leaf attributes of such structures. Then, we can replace all occurrences of the leaf symbols in the generated code with the proper pointer or structure reference syntax to access this leaf attribute.

Despite the extent of the customizations and improvements made over time to Stateflow and to ITLC, we adopted a strategy that excluded all manual editing of the Stateflow and ITLC-generated code. Since our code generation approach is complete in that it targets both the structure and the behavior of the software, there are no other software attributes that require user input. Without manual editing of the generated software, we have been able to focus our efforts on ensuring that the code generators addressed all software integration issues necessary to enable early testing of the whole fault-protection software. As shown in Fig. 5, the integration and test of the whole DS1 avionics software including fault-protection started about six months after start of the FP design effort.

5. TESTING

As is always the case when new technologies are introduced to the conservative environment of interplanetary flight projects, there was substantial discomfort in relying on automatic code generation. Most of the concerns fell into three categories:

1) Is the state diagram format adequate to describe the richness of behaviors required for the fault protection system?
2) Can the code generator be trusted to implement the desired logic in C?
3) Will the FP team end up spending more time debugging the code generator than debugging the flight software?

The FP team's approach for alleviating these concerns was "Testing, testing, and more testing." The plan was to rapidly prototype the most complex behavior required of the DS1 fault protection system and exhaustively test its generated software. The rationale being that if the most complex behavior can be developed using this methodology, the feasibility of developing less elaborate behaviors would no longer be in doubt. The behavior chosen for the prototype was Standby Mode, which is the algorithm entrusted to get the spacecraft power positive, communicative, and thermally safe in case of emergencies.

Testing was conducted in three phases: unit testing, DS1 Testbed testing, and spacecraft system testing. During unit testing, the goal was to test every branch in the fault protection logic in a stand-alone environment. On the DS1 Testbed, the objective was to test only the most likely fault scenarios while including the interactions and timing with other flight software elements, using a copy of the flight computer, and the 1553 bus. Finally, testing on the actual spacecraft concentrated on exercising behaviors that utilized extensive software to hardware interfaces.

Several requirements hinged on the design of the FP unit test platform:

1) Isolation: To remove external dependencies on other flight software modules, all external interfaces to the fault-protection software were completely simulated.
2) Rapid-prototyping: To enable quick turnaround between analysis and design of FP monitors and responses, we needed a mechanism to quickly exercise new designs, to verify that design modifications preserved earlier results, and exercise complex scenarios.
3) Behavior reconstruction: Eventually, testing would move to other platforms were visibility is greatly reduced to the real-time operation of the flight software. To enable detailed analysis of the FP software behavior, we needed to ensure that real-time events recorded in other testbeds could be matched to specific FP scenarios on unit test. Once matched, a non-realtime replay of the scenario on a unit testbed then

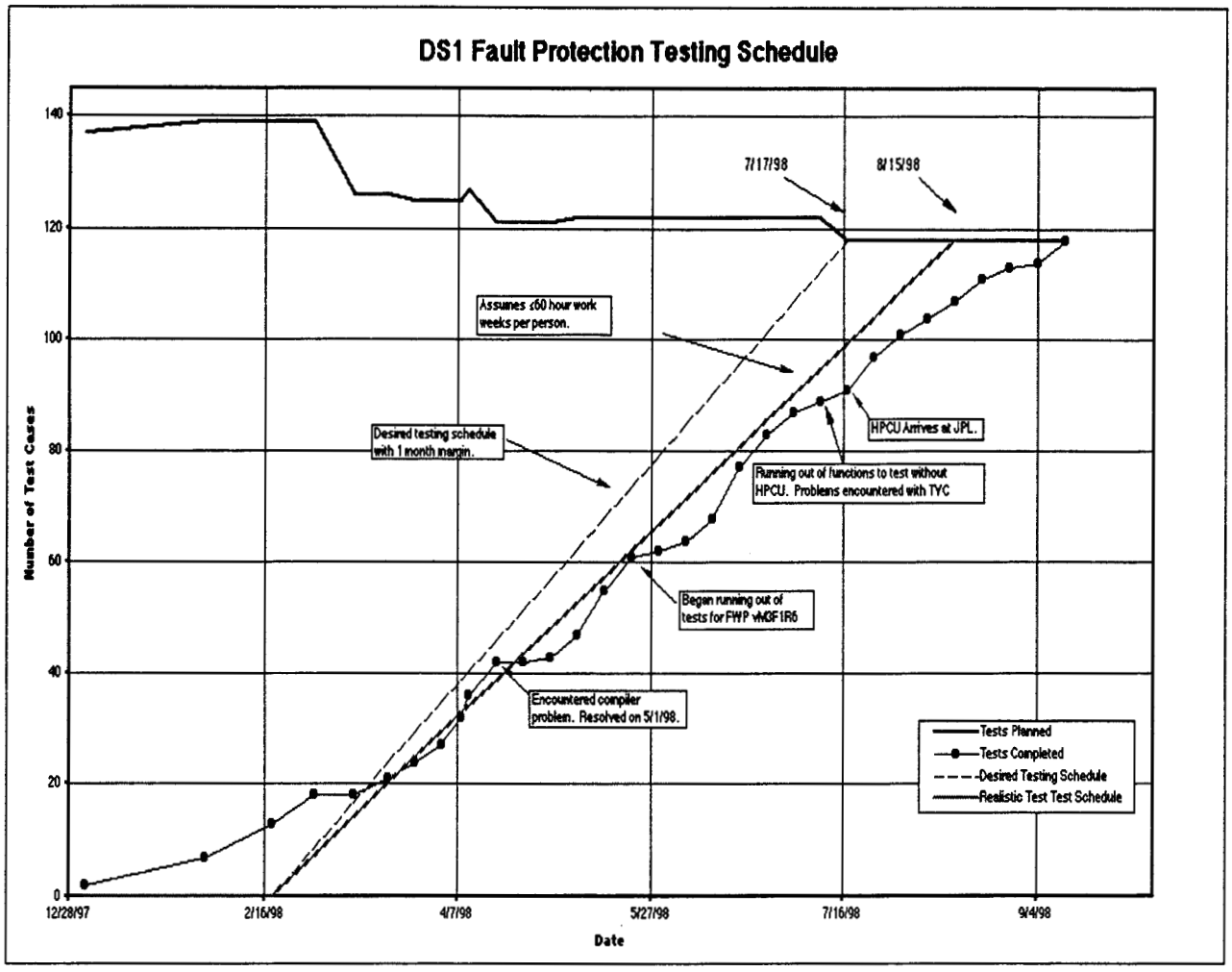


Figure 5: Evolution of the DS1 Fault-Protection scope and progress from design to test.

provided additional visibility into the inner-workings of the FP mechanisms.

The white-box unit tests were performed one or two modules at a time, with very little interaction with other flight software elements and no interaction with actual flight hardware. Each unit test consists of a script of stimulus/response commands. Stimulus commands emulate the state of external interfaces while response commands check the internal and external state of the FP software. The unit tests provided little insight into whether the messages and flags to and from the fault protection system were synchronized with other flight software elements. In spite of these shortcomings, this phase of testing was valuable because the objective was to exercise all paths through the software. Such extensive testing of the logic would not be possible on the DS1 Testbed or on the actual spacecraft due to time constraints. Test time on the DS1 Testbed was an oversubscribed commodity since every subsystem needed it to validate their software as well as the operations team who used it to test flight sequences. Time on the spacecraft was even scarcer since all of flight software testing had to share time with electrical, mechanical fabrication and test activities.

To provide white-box visibility into the FP software, unit test scripts do not run in real-time: the clock can be stopped at any time to analyze the state of the FP monitors and responses. Because all external interfaces are simulated in non-realtime, a unit test exercises only one possible interleaving of the order in which external events and data can occur. This limitation turned out to be helpful to reconstruct specific scenarios seen in other testbeds or the spacecraft. But there is a drawback as well. Since the unit test environment simulates all external interfaces, it follows that external loops between FP and other flight software modules must be closed within the logic of a test script. This part turned out to be one of the most challenging aspects of scripting tests because we did not have a simulation model of the flight software modules at the level of their interfaces, internal state and behavior.

Most likely fault scenarios were tested on the DS1 Testbed. Spacecraft testing concentrated on functions that exercised software to hardware interfaces. About 75% of the testing was completed in test facility in Pasadena. Late software changes were tested at Cape Canaveral. The unit test effort helped the system-level test of the fault protection software in two other ways:

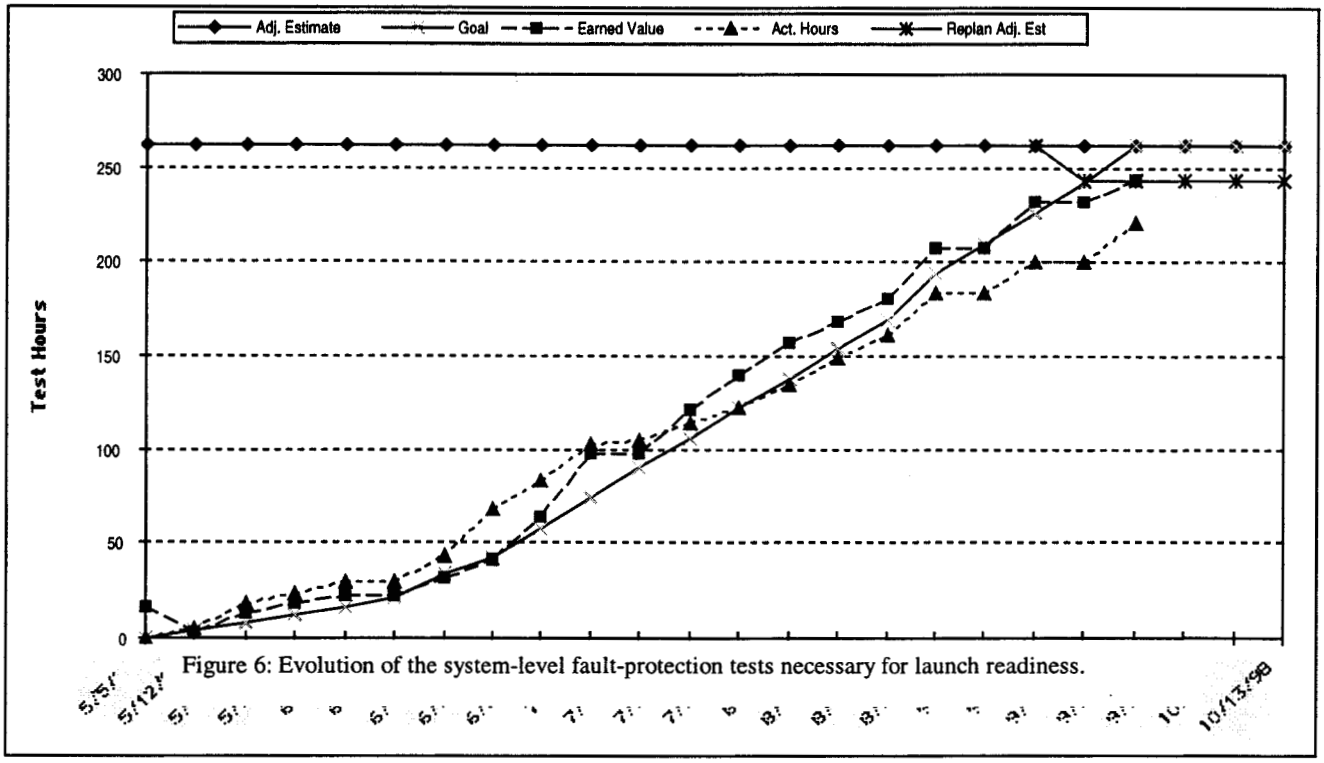


Figure 6: Evolution of the system-level fault-protection tests necessary for launch readiness.

1) To refine the estimates of how many system-level test cases are necessary as a function of the fault-protection scope and
2) Adjust the pace of the design & unit test effort in order to make deliveries of the FP software sufficiently early to start system-level tests.

Figure 5 shows how the scope of the FP design similarly evolved in order to reach a fully-tested, consistent FP system. Figure 6 shows how the scope of the system-level test effort evolved over time.

6. In-Flight Experience

In addition to the traditional fault-protection software, the FP subsystem used the code generator to produce the flight software that would guide the spacecraft through post-separation and initial acquisition activities. Due to power concerns, DS1 was launched with the main processor powered off until spacecraft separation, at which point fault protection software was used to autonomously perform the following functions:

1) Configure the proper state of all the devices, switches, heaters and s/w states.
2) Command the Attitude Control System (ACS) to detumble to rates less than .05 °/hr.
3) Command ACS to acquire a celestial reference and sun point.
4) Deploy the solar arrays by turning on the primary and, if needed, redundant High-Output Paraffin (HOP) deployment actuators and turn them off when finished.
5) Command ACS to re-acquire attitude reference, re-orient to sun and place the panels on sun.
6) Re-configure heaters and configure the telecommunication system for downlink.

Figure 2 is a simplified picture of the main state chart that drove the launch and initial acquisition process. This chart shows the variety of features we used in the state charts: commands or messages to various tasks, messages from various tasks and timers. In the launch configuration state, FP sends commands to an assortment of software tasks or managers to configure for initial acquisition. In the Stellar Reference Unit (SRU) acquisition state, FP commands ACS to estimate the spacecraft attitude and to sun-point, then waits for a message from ACS that it has completed the turn, given up (timed-out) or that Navigation cannot supply ACS with a valid ephemeris. During wing deployment, timers were used to turn on the HOPs for three minutes. Then the FP checks for release or deployment status from the Power subsystem to determine if another attempt should be made.

One important function in this process was the use of "waypoints" to allow fault protection responses to temporarily or permanently interrupt the current response. Waypoints reside in the detumble and SRU acquisition states, at which point, if there is a pending fault protection response, action may be taken.

Our in-flight experience aptly utilized all the state chart features. First of all, the detumble took almost nine minutes, six minutes longer that the expected detumble performance. Luckily, the software was set up to wait for

an ACS message that the rates were low enough to continue (rather than depending on a timer). Next, ACS could not acquire a celestial reference, so after trying five different attitudes, they sent a message to FP that the acquisition attempts timed-out which enabled the transition to the 1st deployment attempt state.

Solar panel deployment occurred much faster than expected and the arrays released 75 seconds after the heaters were turned on and completely latched up 90 seconds after that. This positive release and deployment confirmation enabled the transition to the extra deployment state to allow the wings to fully extend. In doing so, this transition bypassed all of the redundant deployment attempts we had allocated should the sensors have failed for example. The spacecraft essentially remained idle for over four minutes. However, the SRU acquired and began tracking during this lull so ACS immediately started the turn to sun after the deployment timer expired and the FP was reconfigured.

During the second Sun acquisition state, the SRU processor began producing internal checksum errors and illegal software variable values. These events produced a persistent Celestial Inertial Reference Loss (CIRL) condition for the CIRL monitor to declare a fault. Since the 2nd Sun Acquisition State is also a waypoint, the fault-protection engine suspended the launch statechart and started the CIRL response. The FP CIRL response power cycled the SRU, which seemed to clear up the problem after a few minutes and the SRU re-acquired and began tracking. After the turn to sun completed, FP commanded a reconfiguration of the heater states, turned on the power amplifier and initiated X-band downlink.

6. CONCLUSIONS

Until Deep-Space One, automatic code generation technology had not been used on large systems for spacecraft avionics software. Overall, we had a positive experience with this approach on DS-1. The rigorous separation of concerns allowed the fault-protection team to concentrate its efforts on the analysis and design of fault events and associated responses. This approach streamlined interfacing with other teams in the project. The effort spent on tailoring a COTS solution to the needs of the project underline the importance of open code generation algorithms and processes to mitigate the risks inherent in relying on code-generation.

ACKNOWLEDGEMENTS

The authors wish to thank John Slonski for his invaluable contributions to the analysis and design of the DS-1 fault-protection system; Daniel Erickson for his admirable leadership of the DS-1 flight software team; and Dankai Liu for his commendable leadership of the DS-1 avionics team. The authors also wish to thank these three managers for their trust, support and encouragement. This work would not have been possible without the dedicated technical support and assistance of Mehran, Vijay and Jay Thorgenson of the Stateflow team at the Mathworks. In addition, the authors wish to acknowledge all the engineers on the DS-1 team who supportted the development, implementation and testing of this technology. The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Rodney Bell, *Code Generation from Object Models,* http://www.embedded.com/98/9803fe3.htm

[2] Nicolas Rouquette and Daniel Dvorak, *Reduced, Reusable & Reliable Monitor Software,* Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space, 1997.

Nicolas Rouquette is a senior computer scientist and the software engineer for the Deep Space One fault-protection system at the Jet Propulsion Laboratory. He holds a Ph.D. from the University of Southern California in Computer Science.

Tracy Neilson is an avionics systems engineer and fault-protection engineer for the Deep-Space One spacecraft at the Jet Propulsion Laboratory. She hold a B.Sc. from the California State Polytechnic University at Pomona. She was the technical lead engineer for the attitude and articulation control subsystem as well as fault-protection engineer for the Galileo spacecraft.

George Chen is the lead fault-protection engineer for the Deep Space One spacecraft at the Jet Propulsion Laboratory. He holds an M.Sc. from the Massachusetts Institute of Technology in Aerospace Engineering. He was the attitude control engineer for the Mars Observer and Mars Global Surveyor projects.